Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST

Matteo Collina*, Giovanni Emanuele Corazza†, Alessandro Vanelli-Coralli†
*ARCES (Advanced Research Centre for Electronic Systems)
University of Bologna, Viale Risorgimento 2, 40136 Bologna, Italy
Email: mcollina@arces.unibo.it
†DEIS (Department of Electronics, Computer Science, and Systems)
University of Bologna, Viale Risorgimento 2, 40136 Bologna, Italy
Email: {giovanni.corazza, alessandro.vanelli}@unibo.it

***Abstract*—In the "Internet of Things" (IoT) vision the physical world blends with virtual one, while machine-to-machine interaction improve our daily life. Clearly, how these virtual objects are exposed to us is critical, so that their user interface must be designed to support the easiness of usage that is driven by the users' needs, which is different from what machines requires. These two requirements must be solved, and an integrated solution should emerge, if we want to bring the IoT to the 50 billions network that is predicted to became in the next years.**

We believe that these requirements cannot be met by the same communication protocol, and so we propose a new kind of broker, named *QEST* that can bridge the two worlds, represented by their state-of-the-art protocols: MQTT and REST. In this paper, we demonstrate that our approach allows rapid development of user-facing IoT systems, while grating machines all the performance they need.

***Keywords*-Internet of Things, Web of Things, REST, MQTT, smart objects, publish subscribe**

I. Introduction

The last 20 years of research in the pervasive computing area have seen very important steps towards the realization of Mark Weiser's vision of ubiquitous computing [1]: a world were technology vanishes in the background.

The advent of the World Wide Web revolutionized the way we work, communicate, socialize, and how business is done: the Internet has become one the most pervasive technology. Recently, smartphones and mobile broadband enabled us to carry the Internet in our pocket, seamlessy integrating it in our lives. However, everyday objects remain disconnected from the virtual world, while the "Internet of Things" (IoT) movement is exploring how to interconnect them. This technology shift is supposed to be greater than the advent of mobile phones, and in [2] a 2020 scenario where non-phone interconnected devices will be 10 times the phone devices (50 vs 5 billions) is foreseen.

A. Scalability issues

Billions of interconnected devices is a challenge for the whole Internet and for the objects themselves, and in order to operate at that scale several major issues should be resolved. We identified two main scalability issues regarding communication:

1) Firstly, in a several-billions network, a standard, machine-to-machine protocol is needed, so smart object developers could focus on functionalities, and this protocol must be tailored to the machines needs.
2) Secondly, it should be simple for people to interact with machines, usually through their handheld device; the latest years in the mobile industry have shown how the communication should be engineered to support the user experience, and not vice versa.

In general the requirements of machines and people are distinctly different, and it may be hard to define a protocol and the associated best practices to statisfy both of them. We propose a solution that, through the use of widespread patterns in two different areas, can solve the dilemma.

B. A case for the Web of Things

The "Web of Things" (WoT) initiative envisions a world where devices are exposed using the lingua franca of the Internet technologies: the World Wide Web [3]. The WoT architecture builds upon the Representational State Transfer (REST) principles [4]: this paradigm envisions a world where digitally enhanced objects (smart things) are accessible using Uniform Resource Identifiers (URIs) that can be exchanged, referenced and bookmarked. Even further, Guilard et al. [3] propose a way to enable tech-savvy end-users to create physical mashups. This approach enables application developers to leverage their existings skills (HTML, JavaScript, PHP, Ruby, Python) to build new ways to interact with objects , somewhat serendipitously. Semantic discovery services, such as *DiscoWoT* [5], for the Web of Things have also been proposed: thus enabling both humans and machines to semantically discover, select and use smart things.

The WoT as presently conceived, is structured as a huge Service Oriented Architecture (SOA) [6], where each device offers some services to the others. While this approach fits perfectly a business driven use case, it lacks the support needed by the autonomous interaction between smart devices. In [3], the authors noted that in most pervasive scenarios, humans and machines are interested in real-time data, and not in syndication. While the HTTP protocol, which is the underlining medium of the WoT, has not been designed to support persistent communication, the recent draft of HTML 5 specification proposes to introduce *Web Sockets*: a bidirectional communication channel between client and server. However, *Web Sockets* totally hides the naming scheme that makes REST so powerful: every resource has a standard unique identifier, the URI. The *Web Sockets* approach results in non-standard solutions for manipulating resources, whereas, in order to clearly support collaboration between devices, there is the need to unify the naming scheme of smart objects and the URIs, combined with the REST pattern, provide the state-of-art solution.

C. *A case for MQTT*

Today's real world embedded devices usually lacks the ability to handle high-level protocols like HTTP and they may be served better by lightweight binary protocols. In this regard, Andy Stanford-Clark, and Arlen Nipper proposed in 1999 what now has became the MQTT protocol [7].

The MQTT protocol is fast, lightweight, power efficient and implements various levels of Quality of Service. MQTT implements a classic publish/subscribe (pub/sub) pattern with a central broker. The protocol revolves around the concept of *topic*, where clients might publish updates or subscribe for getting the updates from other clients. The MQTT community claims that a pub/sub protocol is what is needed to build a true IoT.

MQTT libraries have been provided for all major IoT development platforms, like Arduino, for several programming languages (C, Java, PHP, Python, Ruby, Javascript) and for the two major mobile platforms (iOS and Android).

The aim of our work is to bring toghether the REST-oriented web architecture with the real-time properties of the MQTT protocol, in order to bridge the gap between machines and developers in the billionaire IoT. We demonstrate that MQTT and REST might and should work together by defining and implementing a new interaction protocol between devices and the Web. The implementation of this protocol is a broker that support the publish/subscribe pattern through multiple communication protocols. We identify our broker as *QEST*, aiming at exposing a MQTT topic as a REST resource.

II. Introducing the QEST broker

The *QEST* broker requires bridging two diverse ecosystems, with a different semantic model of communcation. MQTT implements pub/sub, while the Hyper Text Transfer Protocol (HTTP) - which is the basis of the REST pattern - is just a request/response protocol. *QEST*, by bridging these two approaches, realizes a new hybrid paradigm: firstly it modifies the broker semantic to retain and syndicate the last payload seen on a *topic*, secondly it exposes that payload as a REST resource. In the following paragraphs, we explain

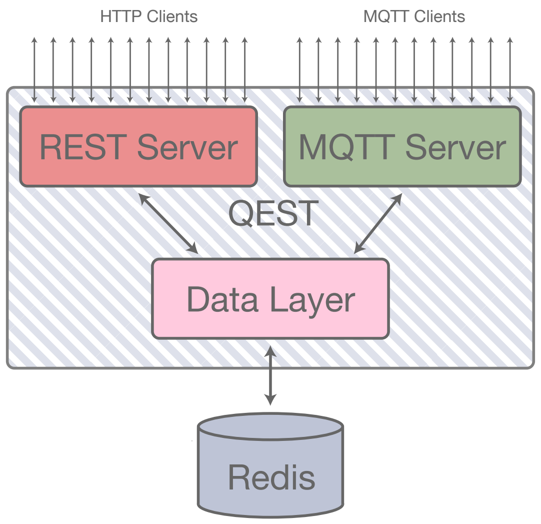

Figure 1. QEST broker architecture

how to achieve this two-way bridging between MQTT and REST.

A. *Exposing MQTT topics as REST resources*

MQTT *topics* do not retain state: however, in order to expose them as a REST resource, we have to syndicate the last payload seen on the *topic* as the resource value: thus, our broker has state. After this consideration, exposing the *topics* is possibile by manipulating a single value. Following a pure REST style, we can create a topic just by issuing an HTTP `POST` request at `/topics` containing both the topic's name and its original payload. For example, if the *topic*'s name is `light_bulb` we can:

- read the last published value by doing an HTTP `GET` request at `/topics/light_bulb`;
- publish a value in the topic by doing an HTTP `PUT` request at `/topics/light_bulb`.

The HTTP protocol allows proper querying of the state of a MQTT topic, because through the use of the HTTP caching protocol, such as the headers `Last-Modified`, `Last-Modified-Since` and `ETag`, clients can safely poll the broker about the state of the *topic*.

This approach can guarantee only a best-effort level of Quality of Service, because that is the true nature of the HTTP protocol.

B. *Exposing REST resources as MQTT topics*

The MQTT protocol is stateless in regard of payloads, but as stated in the previous paragraphs, we need to store the latest published value of a *topic* for syndication. Thus we need to improve the MQTT semantic in order to achieve a better correspondence with that of HTTP.

In MQTT, when a machine subscribes to a *topic*, it does not know the value of the *topic* until a new payload is published. Obviously this is very far from HTTP syndication model, so the broker sends the last published value to the newly connect client. This is an important change because it deviates from the pub/sub nature of the MQTT protocol.

C. Real-time updates for the Browser

Web browsers might want to subscribe for *topic* updates directly on the broker, thus getting near real-time updates. In order to implement this feature using a REST pattern, we propose to use the *Long Polling* approach [8], which allows clients to connect to a resource and wait until it changes or a timeout is passed. As we are adhering to the REST principles, we do not want to change the URI, so the long polling can be triggered by adding a custom "`Long-Polling: enabled`" HTTP header to the GET request.

Instead of using *Long Polling* we could have used *Web Sockets* [9] as the proposals of the WoT suggests [3]. WebSockets however do not implement the concept of URI after opening the communication channel, i.e messages in the stream are free form. The URI is the pillar of the REST architecture, and we deem necessary to adhere to a pure REST approach in regard of the representation of MQTT *topics*.

D. Real-time updates for system integration

The *QEST* broker is not limited at the communication with Web Browser and smart things, but it also supports other back-end systems, such as logistics applications. These are built with procedural or object-oriented paradigms, and they usually require mantaining a connection using threads, or forking processes. It is often quite hard to integrate real-time communication in these legacy applications, as traditional back-end technologies are ill suited for this purpose.

In the previous section we have shown how Web browsers can subscribe to changes using a standard HTTP technique. However, it is hard to implement *Long Polling* in legacy systems, as maintaining an open connection, as *Long Polling* dictates, may be impossible if the technology does not support it. In order to scale the IoT we need simplify the integration with these systems, enabling a communication between them and the *QEST* broker.

In this regard we propose to use webhooks [10], which are a pub/sub implementation using the HTTP medium. Using simple `POST` requests, the client can subscribe to updates on the broker by specifying the URI that will be used to send the updates. The broker assures to call all the subscribed URIs when there is an update to the *topic*.

III. QEST System architecture

QEST revolves around the fast key-value store *Redis* [11], which acts as the internal broker of the application. *Redis* has various features, but the most important for our case is an implementation of the publish/subscribe pattern.

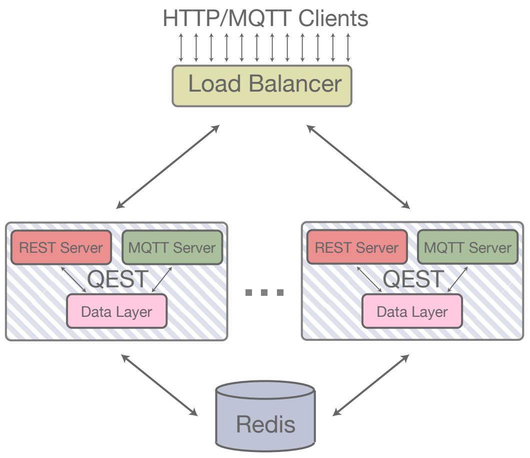

Figure 2. QEST multiple node architecture

As shown in figure 1, a single *QEST* node consists of two main components, the REST (web) server, and the MQTT server: both provide a representation of the data stored in Redis, which is accessed through the data abstraction layer.

QEST is a multiprocol broker with a common semantics, and its main responsibility is to notify subscribers on the update of a particular *topic*, which is achieved as follows:

1) the client publishes the update either through the REST or the MQTT front-end;
2) the front-end writes the new value on the data layer;
3) the data layer stores the value and publishes an update on a specific Redis key.

In order to receive updates on a *topic*, the client acts as follows:

1) the client subscribes for updates to either the REST or the MQTT front-end, according to its nature;
2) the front-end registers for changes to the data layer;
3) the data layer subscribes to changes to a particular Redis key;
4) whenever is published a value on the Redis key, it is forwarded to the data layer;
5) the data layer notifies the subscribed clients through the front-ends.

A. Scalability

In order to support the interaction between billions of devices, the broker must be able to scale horizontally, i.e. the single broker must be replicated mantaining the same functionalities. The *QEST* architecture allows to extend the single node architecture to multiple nodes, as the single node is totally stateless and it can be replicated at will (fig. 2).

The multiple node architecture introduces a load balancer, which is necessary to divide the load between all nodes and it can be implemented using several software components, such as HAProxy [12].

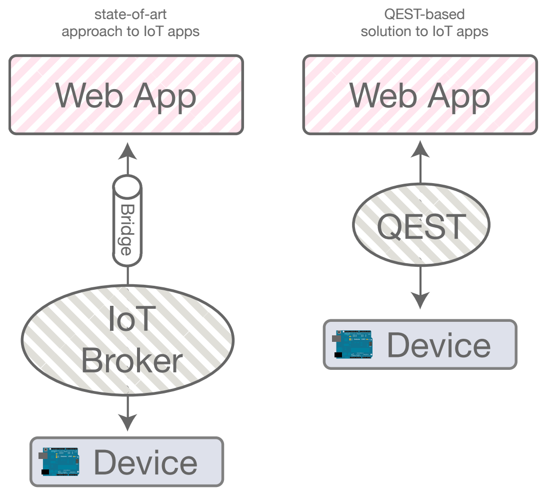
state-of-art approach to IoT apps
Web App
Bridge
IoT Broker
Device
QEST-based solution to IoT apps
Web App
QEST
Device

Figure 3. Example architectures for IoT applications, implemented with the current state-of-art (on the left), or with *QEST* (on the right)

B. Enabling technologies

The principal technology requirement for the *QEST* broker is the ability to handle thousands of concurrent connections with a low overhead. Traditional server technologies use a fork or threading approach, where a process or a thread serve only one request at a time. Clearly this approach cannot support thousands of concurrent connections

A solution for handling this case is to use evented input/output capabilities, where there is a single process/thread that handle each connection when it needs to receive or send data on it. This approach allows a lower memory footprint and low CPU usage; thus is scalable.

In particular, the *QEST* broker is built on the Node.js [13] platform, which is based upon the Chrome JavaScript virtual machine. Node.js implements the reactor pattern: a particular evented I/O system where every computation is executed in response to an event, and this approach allows to build highly concurrent network applications. In order to support the MQTT protocol, we used the MQTT.js implementation [14].

IV. Use Cases

The primary objective of *QEST* is simplifying the development of IoT applications, by allowing the developers to integrate real world devices easily in web and mobile applications. In figure 3, we compare the state-of-art approach and our *QEST* proposal. In the first one, we have to implement a custom bridge between the web application and the broker, which limits the diffusion of the interaction between the real and virtual worlds. On the contrary, *QEST* simplify the solution by concentrating two functions on the same component. In order to demonstrate this approach, we developed two main use cases: ”forest of leds” and ”temp monitoring”.

In the first scenario, we crafted a set of Arduino [15] devices that switches on and off a led based on the state of a common *topic*, which is modified by pressing a button on one of the devices. This allows us to trigger on and off all the leds in our ”forest” by pushing the button of one of the devices, or just from our web interface.

In the second scenario, we built a do-it-yourself temperature monitoring for the house, and exposed the value on a web page. Using this information, we were able to track and graph the temperature of the house.

V. Experimental results

The main point of evaluation of our solution is the easiness of usage for software developers. Also, in order to show that *QEST* does not penalize the performance[1]we compared our solution to two implementations of MQTT: the *Really Small Message Broker (RSMB)* [16], which is a proprietary implementation made by IBM, and *Mosquitto* [17], which is open source.

We compared the performance of raw publishing and receiving a notification on multiple clients, as we believe it is the most challenging for a MQTT broker and hence for *QEST*. As shown in figure 4, QEST performance is comparable with that of *Mosquitto* in the considered interval (1-1000 subscribers). The RMSB implementation performs better up to 100 subscribers, while it does not support a larger number of users. Finally, the two nodes *QEST* setup is faster than the single one, as expected.

We also benchmarked the *QEST* broker HTTP performance on publishing to a *topic* with a growing number of subscribers (fig. 5).

Results demonstrate that publishing a value on the *QEST* broker through HTTP is equally performant to MQTT.

VI. Security issues

In the future, the need to secure communication between things and humans, with a billion devices IOT, is one of the biggest challenges we will have to cope with, to avoid a massive privacy issue. In this regard, the European Union is investigating how to regulate the future Internet of Things [18]. The state-of-art in M2M protocols does not address the security and privacy needs of our society [19], and MQTT makes no exception.

In order to protect a resource, a system should adhere to two different requirements:

- authentication, i.e. allowing only known user to connect to the system;
- authorization, i.e. denying the users the right to access the resource they are forbidden to access.

The MQTT protocol was not designed with security in mind, and we highlight problems in both areas. Regarding

[1]All benchmarks where done with a PC running Ubuntu Linux and equipped with an Intel i5-2400 processor (6MB Cache, 3.10 GHz) and 4GB of RAM.

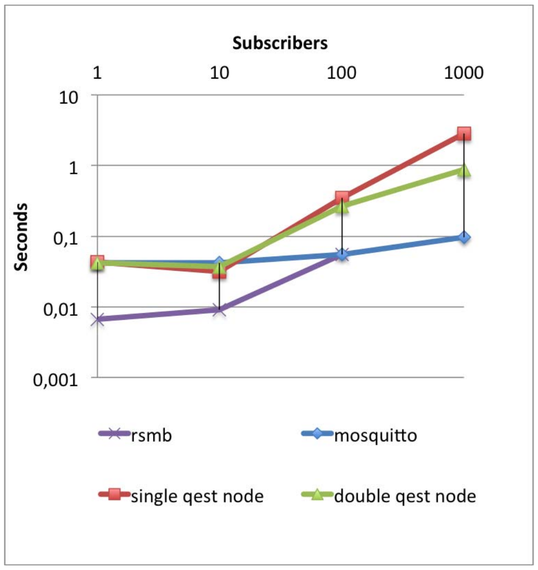

Figure 4. Comparative benchmark on MQTT performance between *Mosquitto*, *RSMB* and *QEST*

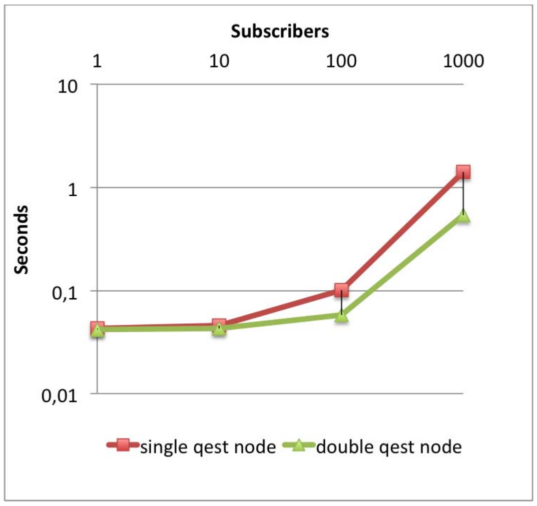

Figure 5. *QEST* benchmark on publishing a value on a *topic* from HTTP, and listening to it from a growing number of MQTT subscribers.

the first one, even though it support username and password based authentication, it transmits credentials without encryption.

In [19], the authors cite Virtual Private Networks (VPN) as a mean to secure IoT deployments, and we propose it too.

As for authorization, all the main MQTT implementations support Access Control Lists (ACL) for limiting users to access resources, but they cannot be updated in real-time, and from the protocol itself. If the IoT system is going to became ubiquitous and consumer-friendly, then an authorization protocol similar to OAuth [20], which is used on the web to grant access to personal data by third-party applications, must be designed and developed. Developing such a solution is out of the scope of our current work, but we believe a mixed MQTT/REST solution might allow the user to pair their devices with their data.

VII. Conclusions and Future Work

In a billion-devices Internet of Things, there is the need of a common medium to support the interaction between machines, and to allow new form of applications between smart objects and end users. We discussed the requirements for the two types of interactions, and we concluded that the state-of-art of communication protocols does not support all of them.

In order to solve the dilemma, we introduced a new broker, *QEST*, that bridge the gap between the things and the web, allowing existing developers to use their skills to interact with smart objects, thus fostering innovation. We demonstrated that *QEST* performance are comparable to other MQTT implementation, while providing a much more efficient programming interface between things and developers.

As for future developments, we plan to expand the *QEST* broker to fully support the MQTT specification, e.g. it only implements best-effort quality of service and, it lacks some peer-to-peer capabilities which are present in other implementations.

Finally, it is possible to extend the *QEST* broker to support sensor networks, i.e. devices that are too much energy efficient to support the permanent TCP connection needed by TCP, which is the basis of the MQTT protocol. Thanks to the multi-protocol architecture of our broker we plan to integrate the MQTT-S protocol, which has been designed to support such devices.

References

[1] M. Weiser, "The computer for the 21st century," *Scientific American*, Feb. 1991. [Online]. Available: http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html

[2] "More than 50 billion connected devices," Ericsson, February 2011. [Online]. Available: http://www.ericsson.com/res/docs/whitepapers/wp-50-billions.pdf

[3] D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the web of things," in *Proc. Internet of Things (IOT)*, Tokyo, Japan, December 2010, pp. 1–8.

[4] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Trans. Internet Technol.*, pp. 115–150, May 2002.

[5] S. Mayer and D. Guinard, "An extensible discovery service for smart things," in *Proceedings of the 2nd International Workshop on the Web of Things (WoT 2011)*. San Francisco, USA: ACM, Jun. 2011.

[6] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services," *IEEE Transactions on Services Computing*, vol. 3, no. 3, pp. 223–235, 2010.

[7] "MQ Telemetry Transport," April 2012. [Online]. Available: http://mqtt.org

[8] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins, "Known issues and best practices for the use of long polling and streaming in bidirectional http. internet engineering task forceprincipled design of the modern web architecture," RFC 6202, April 2011.

[9] "The WebSocket API," World Wide Web Consortium (W3C), April 2012. [Online]. Available: http://dev.w3.org/html5/websockets/

[10] "Webhooks," April 2012. [Online]. Available: http://www.webhooks.org

[11] S. Sanfilippo, "Redis," April 2012. [Online]. Available: http://redis.io

[12] "HAProxy," April 2012. [Online]. Available: http://haproxy.1wt.eu

[13] "Node.js," Joyent, Inc, April 2012. [Online]. Available: http://nodejs.org

[14] A. Rudd, "MQTT.js," April 2012. [Online]. Available: http://bit.ly/hO9cZP

[15] M. Banzi, D. Cuartielles, T. Igoe, G. Martino, and D. Mellis, "Arduino," April 2012. [Online]. Available: http://arduino.cc/

[16] "Really Small Message Broker," April 2012. [Online]. Available: http://ibm.co/GQ7vwr

[17] "Mosquitto," April 2012. [Online]. Available: http://mosquitto.org

[18] "EU investigating IoT regulations," April 2012. [Online]. Available: http://bit.ly/HyYSb2

[19] D. Giusto, A. Iera, G. Morabito, L. Atzori, C. M. Medaglia, and A. Serbanati, "An overview of privacy and security issues in the internet of things," in *The Internet of Things*. Springer New York, 2010, pp. 389–395.

[20] "The OAuth 2.0 Authorization Protocol," April 2012. [Online]. Available: http://datatracker.ietf.org/doc/draft-ietf-oauth-v2